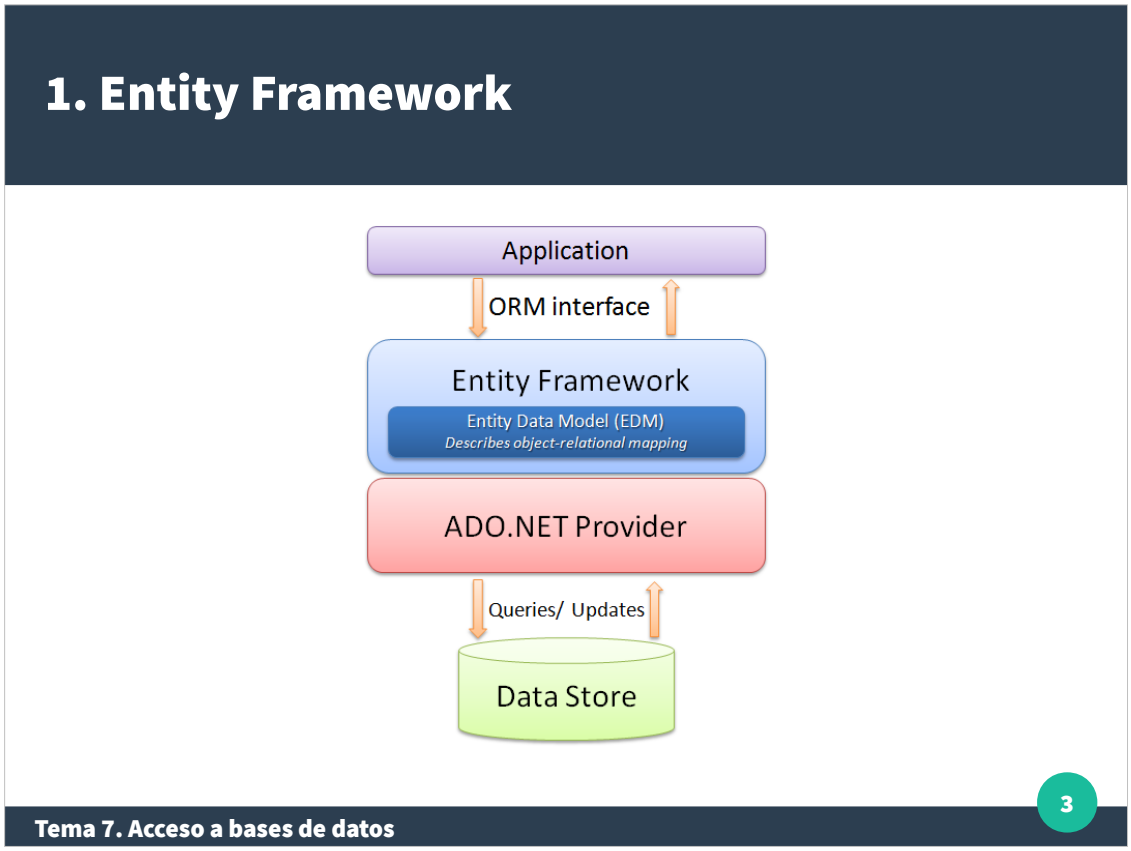
Tema 7. Acceso a bases de datos

Desarrollo de Interfaces
DAM – IES Doctor Balmis

Javier Catalá

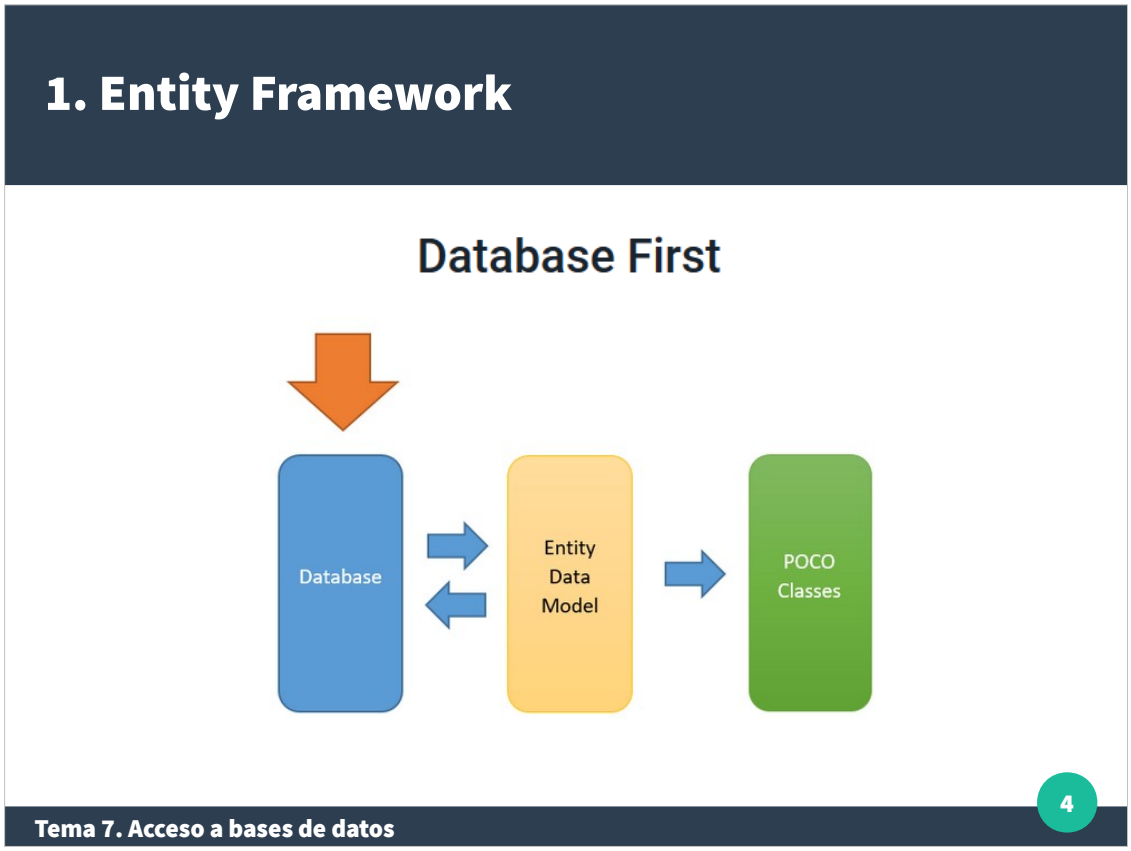
INDICE



En este tema vamos a ver cómo acceder a bases de datos relacionales desde una aplicación WPF utilizando el ORM de Microsoft, *EntityFramework*.

La aplicación interactuará con la base de datos utilizando un conjunto de clases que representarán las diferentes tablas. La relación entre las clases y las tablas se almacena en el modelo (EDM).

Para realizar el acceso efectivo a la base de datos relacional, *Entity Framework* utiliza la tecnología ADO.NET. Si quisieramos acceder desde nuestra aplicación a una base de datos relacional directamente, sin utilizar un ORM, tendríamos que utilizar las clases de ADO.NET.

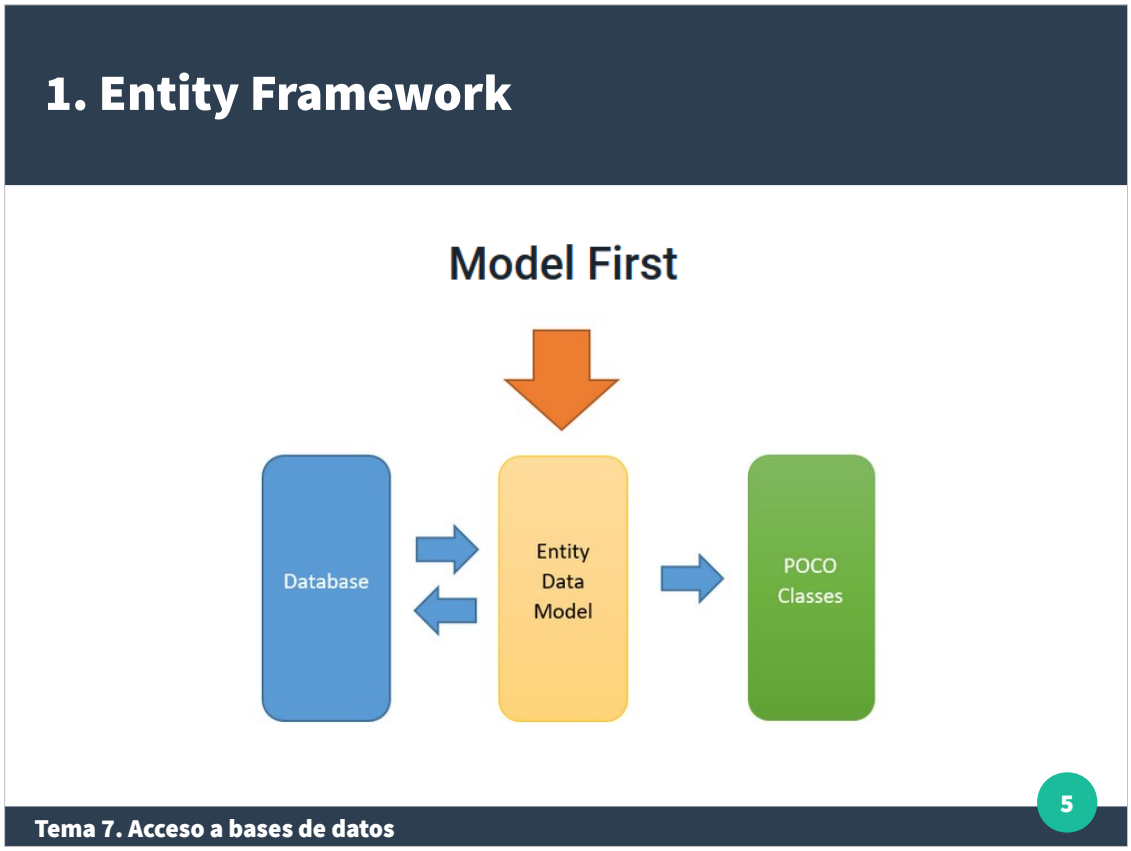


Entity framework dispone de diferentes modos de funcionamiento. Uno de ellos es *Database First*.

En este modo, se comienza con una base de datos relacional existente, y a partir de ella se genera de forma automática el modelo. Futuros cambios en el modelo pueden ser traducidos de forma automática en cambios en la estructura de la base de datos.

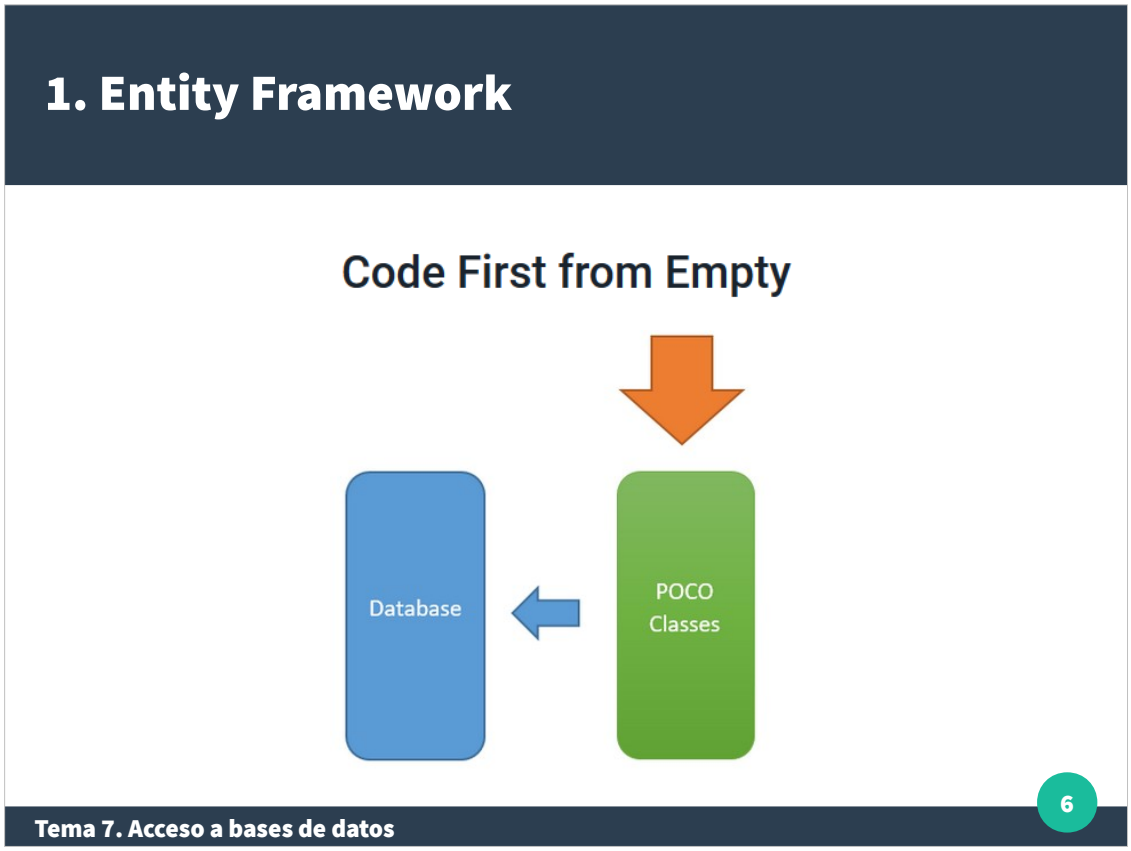
A partir del modelo, se generarán las clases POCO (*Plain Old CLR Object*) que utilizará la aplicación.

Este es el modo que utilizaremos nosotros en clase.



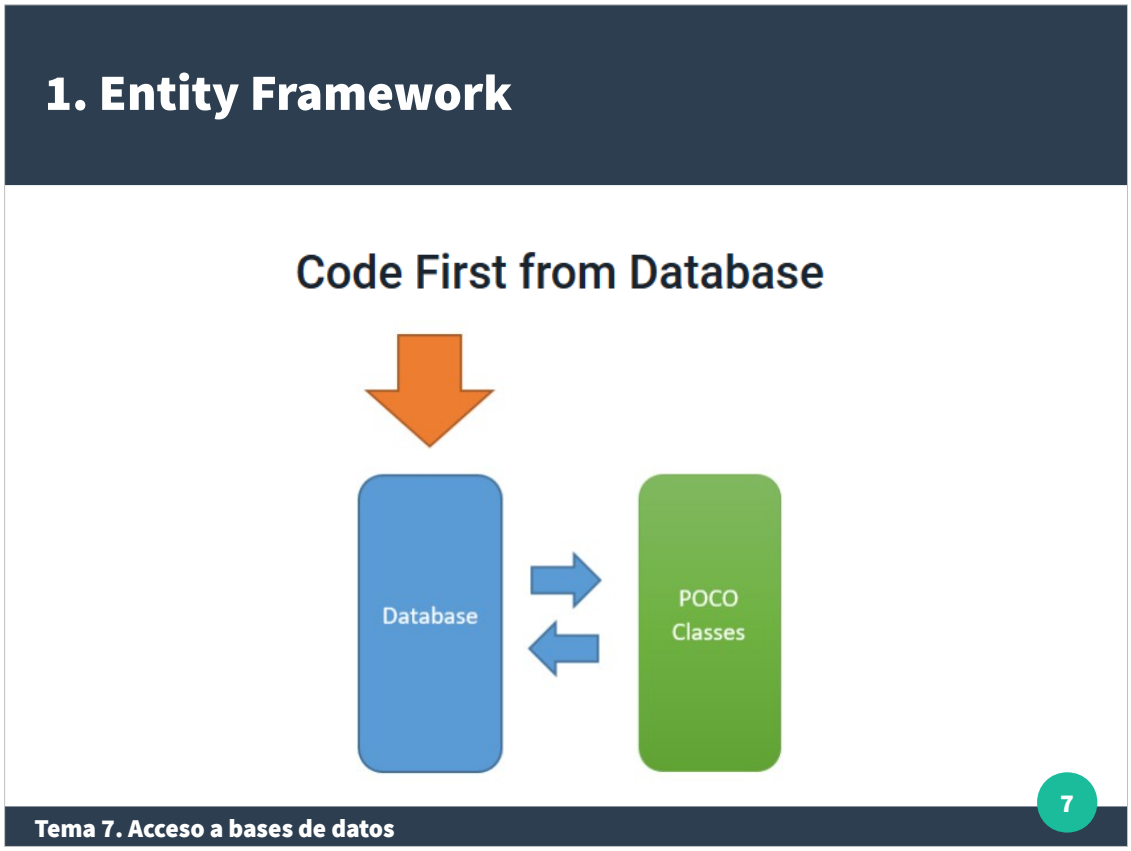
Otra opción es comenzar por el diseño del modelo, y a partir de él generar de forma automática la base de datos y las clases POCO.

El modelo está almacenado en un fichero CSDL (lenguaje de definición de esquemas conceptuales) aunque Visual Studio dispone de herramientas visuales de diseño del modelo.



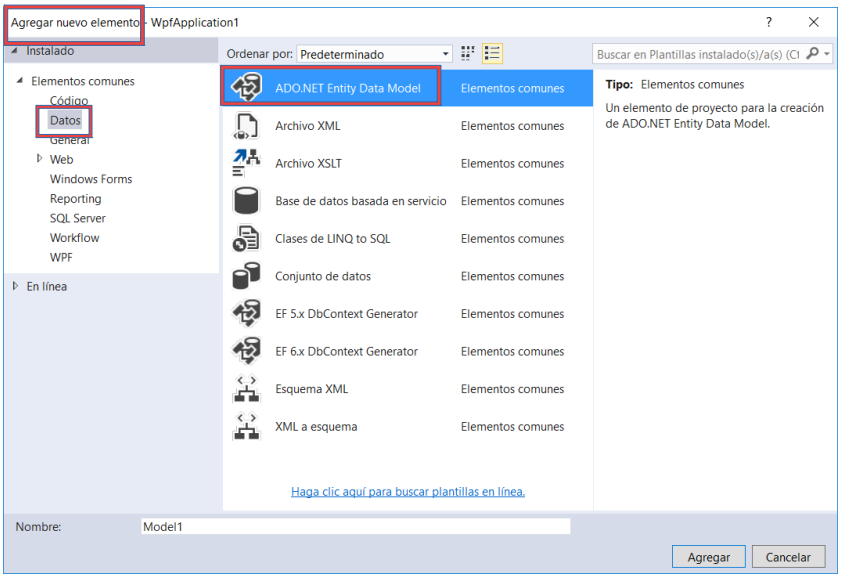
El último modo de funcionamiento de *Entity Framework* se denomina *Code First*. En este modo, se prescinde del modelo y se produce una correspondencia directa entre las clases POCO y la base de datos.

Existen dos posibilidades dentro de *Code Fisrt*. La que se muestra en esta diapositiva es *Code First from Empty*, en la que se comienza con la creación de las clases POCO, y a partir de ellas se generá la base de datos.



La otra posibilidad es *Code Forst from Database*. Partimos de una base de datos existente, y a partir de ella se generan las clases POCO, sin necesidad de generar el modelo intermedio.

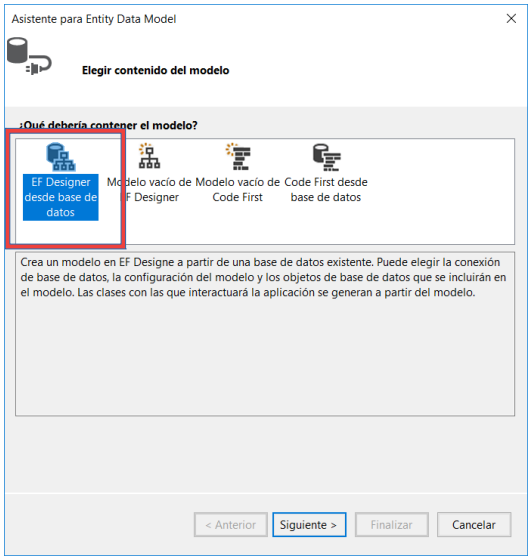
2. Database First



Vamos a comenzar a ver cómo podemos utilizar el modo *Database First* en nuestras aplicaciones.

Lo primero será agregar un nuevo elemento en nuestro proyecto de tipo *ADO.NET Entity Data Model*.

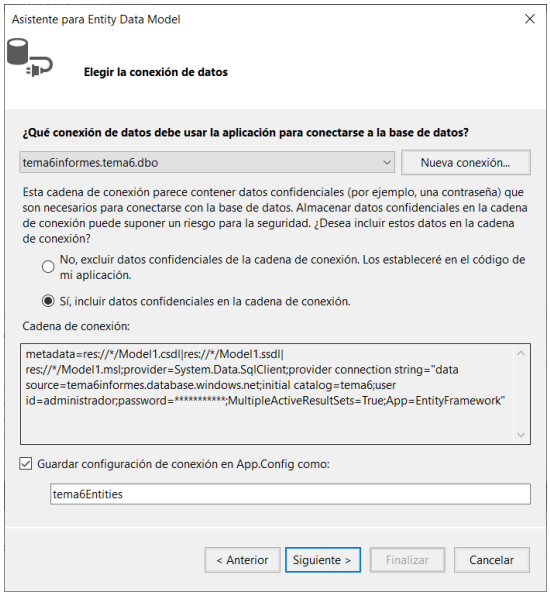
2. Database First



A continuación comenzará el asistente de Entity Framework. El primer paso es elegir el modo de funcionamiento. Como podemos ver en la imagen se ofrecen los cuatro modos que hemos comentado con anterioridad.

En nuestro caso seleccionaremos la primera opción, *EF Designer desde base de datos*.

2. Database First

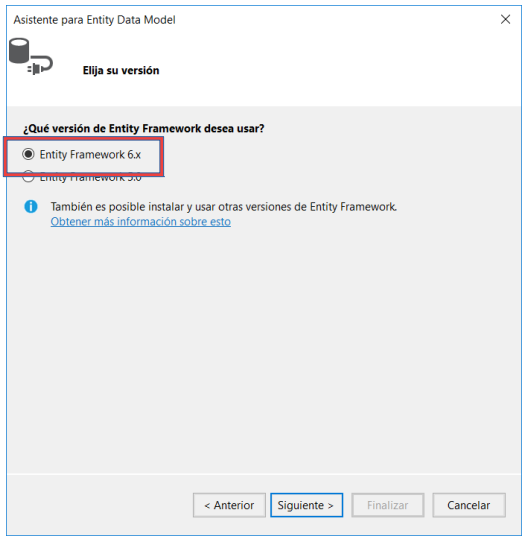


El siguiente paso será establecer la conexión con la base de datos de origen. Dependiendo del tipo de base de datos que utilicemos la información necesaria en este paso será diferente.

Aunque *Entity Framework* se puede utilizar con cualquier sistema gestor de bases de datos relacionales (como Oracle o MySQL) la configuración es más sencilla si se utiliza SQL Server o Azure SQL.

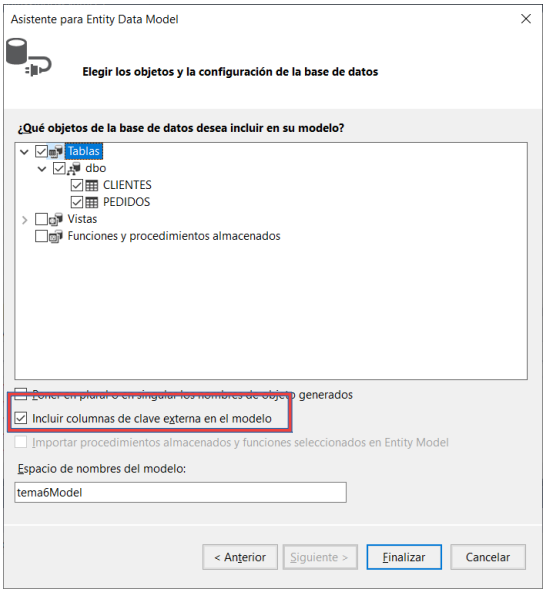
En este paso también podemos indicar que la cadena de conexión se almacene en la configuración del proyecto (recomendable).

2. Database First



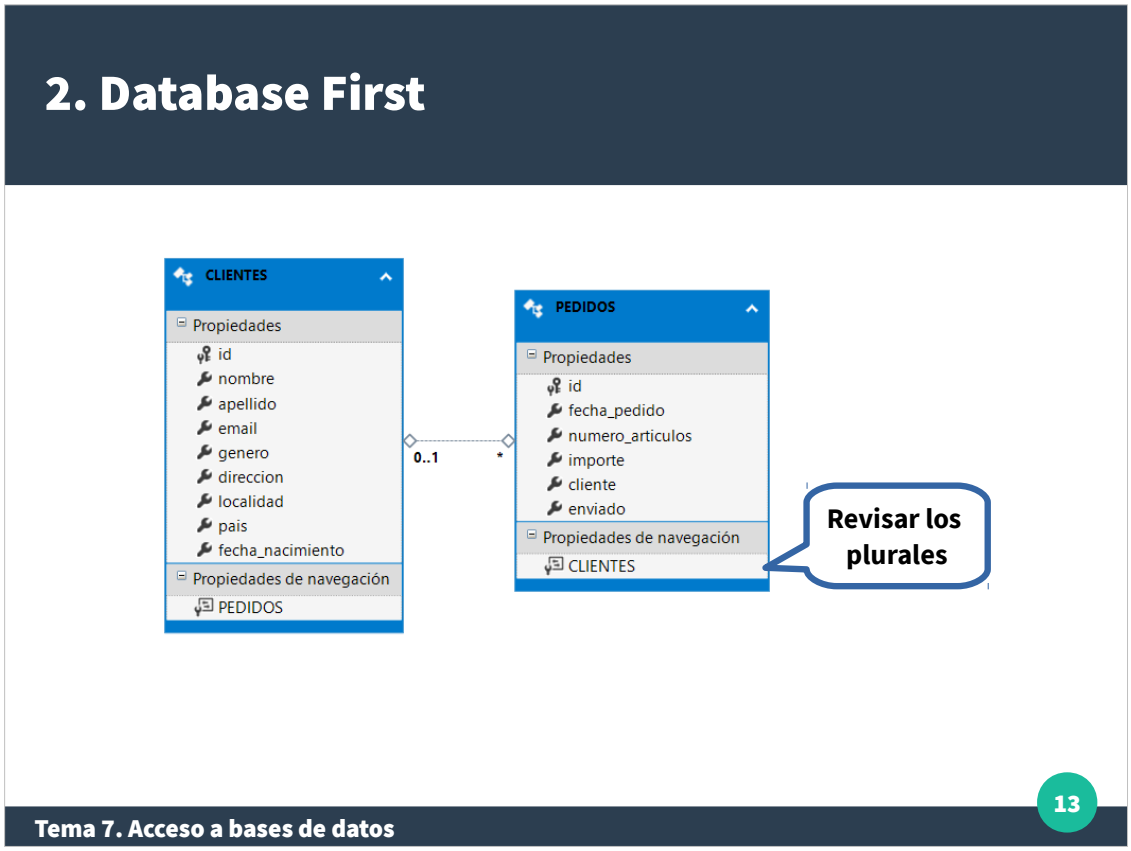
El siguiente paso del asistente será elegir la versión de *Entity Framework*. En nuestro caso, la 6.X.

2. Database First



A continuación, seleccionaremos los objetos de base de datos que queremos incluir en el modelo. Normalmente serán tablas, pero también podríamos incluir vistas o procedimientos almacenados.

En este paso es importante que marquemos la opción *Incluir columnas de clave externa en el modelo*. Esta opción nos permitirá relacionar desde el código objetos de tablas diferentes entre las que exista un vínculo en la base de datos.

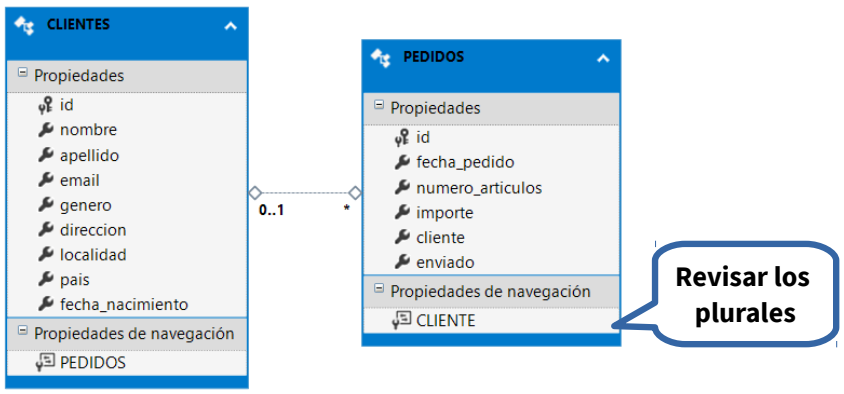


Una vez finalizado el asistente aparecerá el modelo generado en el diseñador de modelos de Visual Studio, con las entidades asociadas a las tablas de la base de datos.

Como podemos observar, además de las propiedades asociadas a las columnas de las tablas se han generado un tipo especial de propiedades llamadas *propiedades de navegación*. Estas propiedades van a permitirnos acceder desde el código a los registros relacionados en otras tablas.

En nuestro ejemplo, la propiedad PEDIDOS de la entidad CLIENTES permitirá acceder a todos los pedidos de un cliente. Y la propiedad CLIENTE de la entidad PEDIDOS permitirá acceder al cliente asociado a un pedido.

2. Database First



Es importante revisar los nombres asignados a las propiedades de navegación, ya que en ocasiones tendremos que hacer algún ajuste referente al uso del plural.

En nuestro caso, la propiedad de navegación presente en la entidad PEDIDOS se llama de forma predeterminada CLIENTES (como la entidad a la que referencia). Sin embargo, un determinado pedido solo puede tener asociado un cliente, por lo que tiene más sentido que dicha propiedad aparezca en singular.

3. Uso de las clases POCO

```
//Instanciamos el contexto para acceder a la base de datos
tema6Entities contexto = new tema6Entities();

//Cargamos desde la base de datos los datos de una tabla
contexto.CLIENTES.Load();

//La propiedad Local proporciona una ObservableCollection con
//los registros de la tabla
ClientesListBox.DataContext = contexto.CLIENTES.Local;

//También podemos utilizar LINQ sobre el contexto
var consulta = from n in contexto.CLIENTES
               where n.genero == "Male"
               orderby n.nombre
               select n;
ClientesListBox.DataContext = new ObservableCollection<CLIENTES>(consulta.ToList());
```


Una vez creado el modelo, para poder interactuar con la base de datos necesitamos un objeto especial llamado contexto. El contexto nos permitirá acceder a las diferentes entidades existentes en el modelo.

En la diapositiva se muestra cómo acceder a los datos de una entidad y utilizarlos como *DataContext* de un *ListBox*. Podemos acceder a todos los datos de la tabla, o utilizar LINQ para hacer un filtrado en los datos.

3. Uso de las clases POCO

```
//Creamos un nuevo cliente
CLIENTES nuevo = new CLIENTES
{
    id = 200,
    nombre = "Javier"
};

//Lo añadimos al contexto
contexto.CLIENTES.Add(nuevo);

//Trasladamos los cambios a la base de datos
contexto.SaveChanges();
```


También podemos utilizar el contexto para añadir nuevos registros a la base de datos.

Para ello, crearemos un nuevo objeto de la clase correspondiente y lo añadiremos al contexto.

Para que los cambios se trasladen a la base de datos real, es necesario invocar el método *SaveChanges* del contexto.

3. Uso de las clases POCO

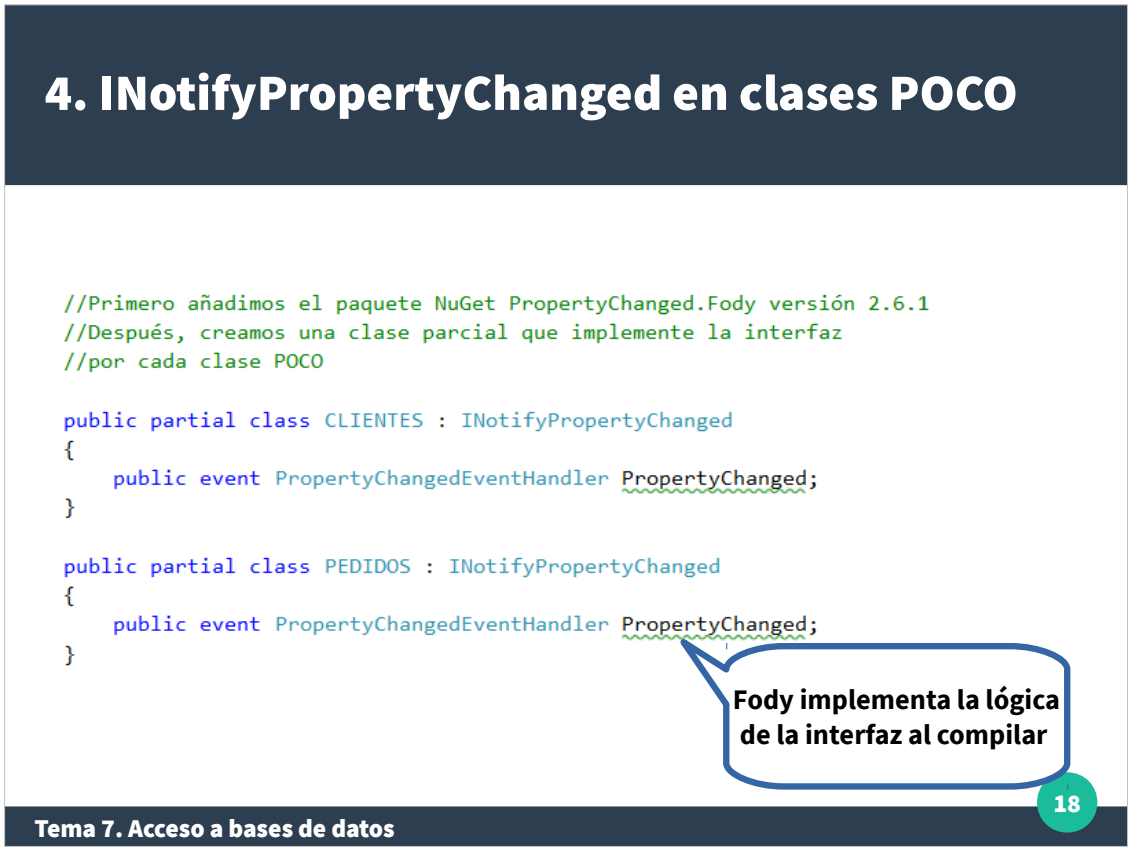
```
//Podemos modificar un cliente
CLIENTES cliente = (CLIENTES)ClientesListBox.SelectedItem;
cliente.nombre = "Pedro";

//O eliminarlo
contexto.CLIENTES.Remove((CLIENTES)ClientesListBox.SelectedItem);

//Trasladamos los cambios a la base de datos
contexto.SaveChanges();
```


También podemos modificar los datos de un registro directamente en el contexto, o eliminar un registro existente.

Todos estos cambios, como ya se ha comentado, no se trasladarán a la base de datos real hasta que se realice la llamada al método *SaveChanges*.



En ocasiones, puede ser necesario que las clases generadas por Entity Framework implementen la interfaz *INotifyPropertyChanged*. El problema de implementar esta interfaz como hemos hecho hasta ahora es que si cambiamos el modelo y se regeneran las clases, tendremos que volver a realizar la implementación.

Para evitarlo, vamos a usar un paquete NuGet llamado *PropertyChanged.Fody*, que implementa la interfaz de forma automática. Lo que haremos será crear una clase parcial con el mismo nombre que la clase generada por Entity Framework, y ahí indicaremos que se implementa la interfaz y añadiremos el evento que ésta indica. Solo con esto, conseguiremos que la clase implemente correctamente la interfaz.

5. El control DataGrid

```
<DataGrid x:Name="PedidosDataGrid" ItemsSource="{Binding}" />
```

```
contexto = new tema6Entities();
contexto.PEDIDOS.Load();
PedidosDataGrid.DataContext = contexto.PEDIDOS.Local;
```

id	fechapedido	numeroarticulos	importe	cliente	enviado	CLIENTES
1	8/10/2019 12:00:00 AM	2	301.42	100	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
2	4/24/2019 12:00:00 AM	3	317.22	97	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
3	10/5/2019 12:00:00 AM	5	415.80	100	1	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
4	10/30/2019 12:00:00 AM	5	67.21	89	1	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
5	5/19/2019 12:00:00 AM	7	276.57	55	1	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
6	5/4/2019 12:00:00 AM	5	452.44	23	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
7	3/29/2019 12:00:00 AM	10	305.90	38	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
8	8/13/2019 12:00:00 AM	4	445.81	67	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
9	1/24/2019 12:00:00 AM	10	387.02	26	1	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
10	7/28/2019 12:00:00 AM	8	203.87	42	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
11	6/9/2019 12:00:00 AM	4	489.02	47	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
12	10/7/2019 12:00:00 AM	6	225.03	13	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA
13	1/31/2019 12:00:00 AM	1	292.00	54	0	System.Data.Entity.DynamicProxies.CLIENTES_DE959B21B35294BA

Existe un control de usuario en WPF especialmente pensado para mostrar al usuario datos en forma de tabla: el control *DataGrid*.

En el ejemplo podemos ver lo sencillo que resulta enlazar un control *DataGrid* a la *ObservableCollection* obtenida con Entity Framework. Para ello, utilizamos la propiedad *ItemsSource* del *DataGrid* (de forma análoga a como hacemos con los controles de lista).

Como podemos ver, en el *DataGrid* resultado se incluyen todas las columnas de la tabla, y una columna más por cada propiedad de navegación, que por defecto muestra el *ToString* del objeto enlazado.

5. El control DataGrid

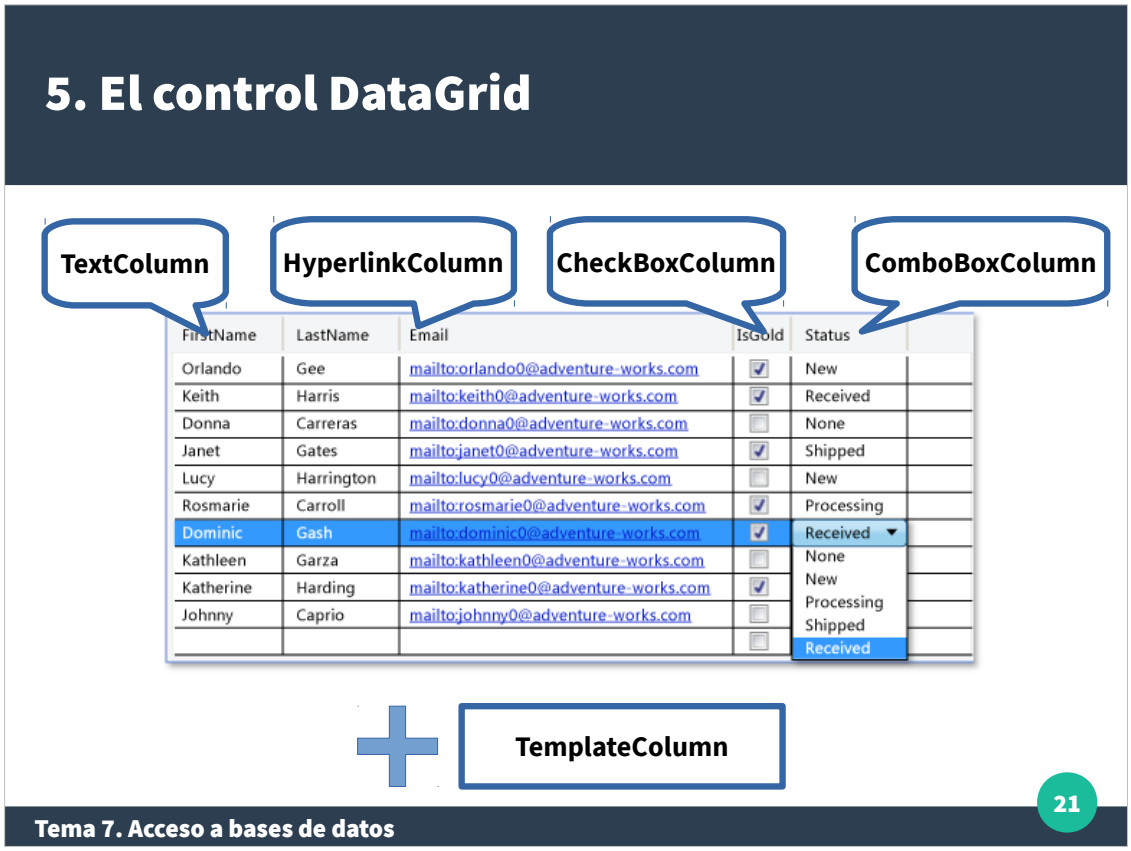
```
<DataGrid x:Name="PedidosDataGrid" ItemsSource="{Binding}" AutoGenerateColumns="False">
    <DataGrid.Columns>
        <DataGridTextColumn Header="Identificador" Binding="{Binding id}"/>
        <DataGridTextColumn Header="Nº Artículos" Binding="{Binding numero_articulos}"/>
        <DataGridTextColumn Header="Importe" Binding="{Binding importe}"/>
        <DataGridTextColumn Header="Cliente" Binding="{Binding CLIENTES.nombre}"/>
    </DataGrid.Columns>
</DataGrid>
```

Identificador	Nº Artículos	Importe	Cliente
1	2	301.42	Karlan
2	3	317.22	Abigail
3	5	415.80	Karlan
4	5	67.21	Wadsworth
5	7	276.57	Estell
6	5	452.44	Abbi
7	10	305.90	Eleanor
8	4	445.81	Philipa
9	10	387.02	Brear

Si queremos personalizar las columnas del *DataGrid* deberemos establecer a false su propiedad *AutoGenerateColumns*, de forma que no se incluyan todas las columnas de la tabla.

A continuación estableceremos las columnas que queremos ver con la propiedad *Columns* del *DataGrid*. Para cada columna, podemos indicar su cabecera con la propiedad *Header*, y el campo a mostrar con la propiedad *Binding*.

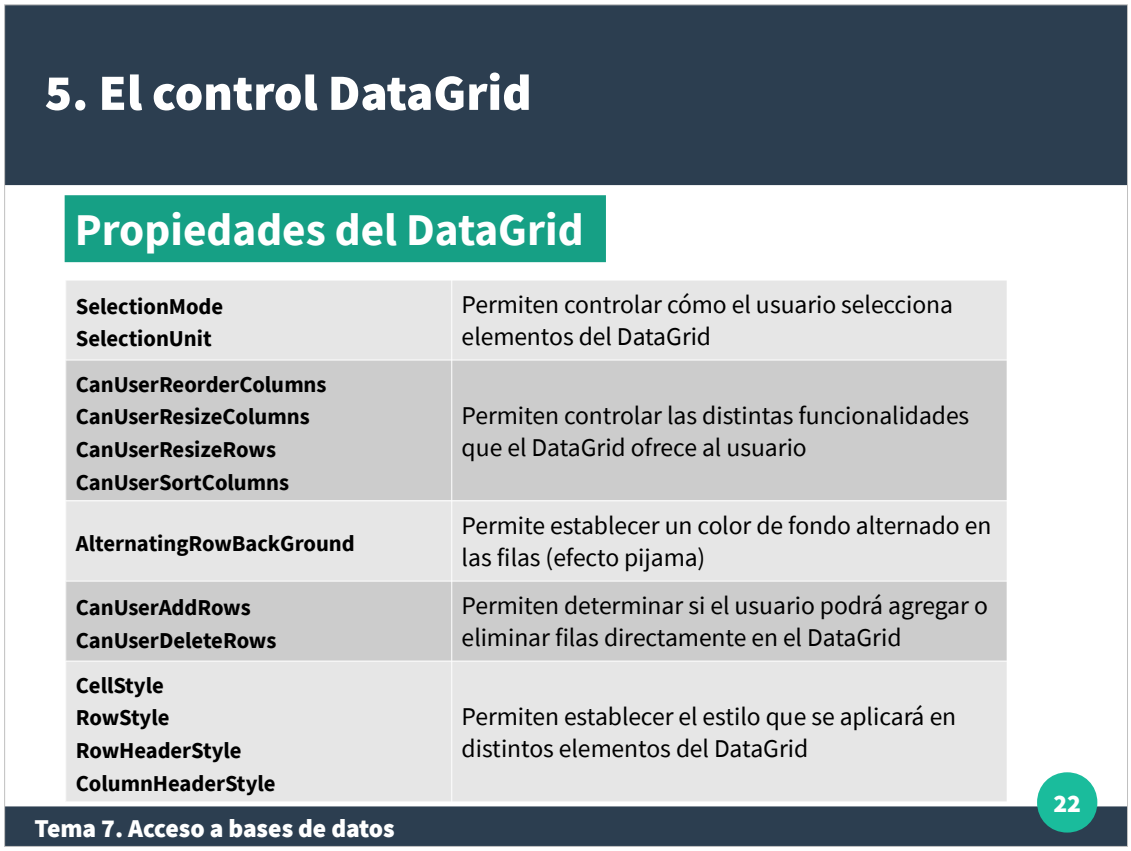
Como podemos ver en la columna que hace referencia al cliente, podemos acceder a los campos de las tablas enlazadas gracias a las propiedades de navegación.



En el ejemplo anterior hemos definido columnas de tipo texto (*DataGridTextColumn*). Sin embargo, el control *DataGrid* permite definir otros tipos de columnas, como puede apreciarse en la dispositiva:

- *HyperlinkColumn*, para enlaces.
- *CheckBoxColumn*, para casillas de verificación.
- *ComboBoxColumn*, para listas desplegables.

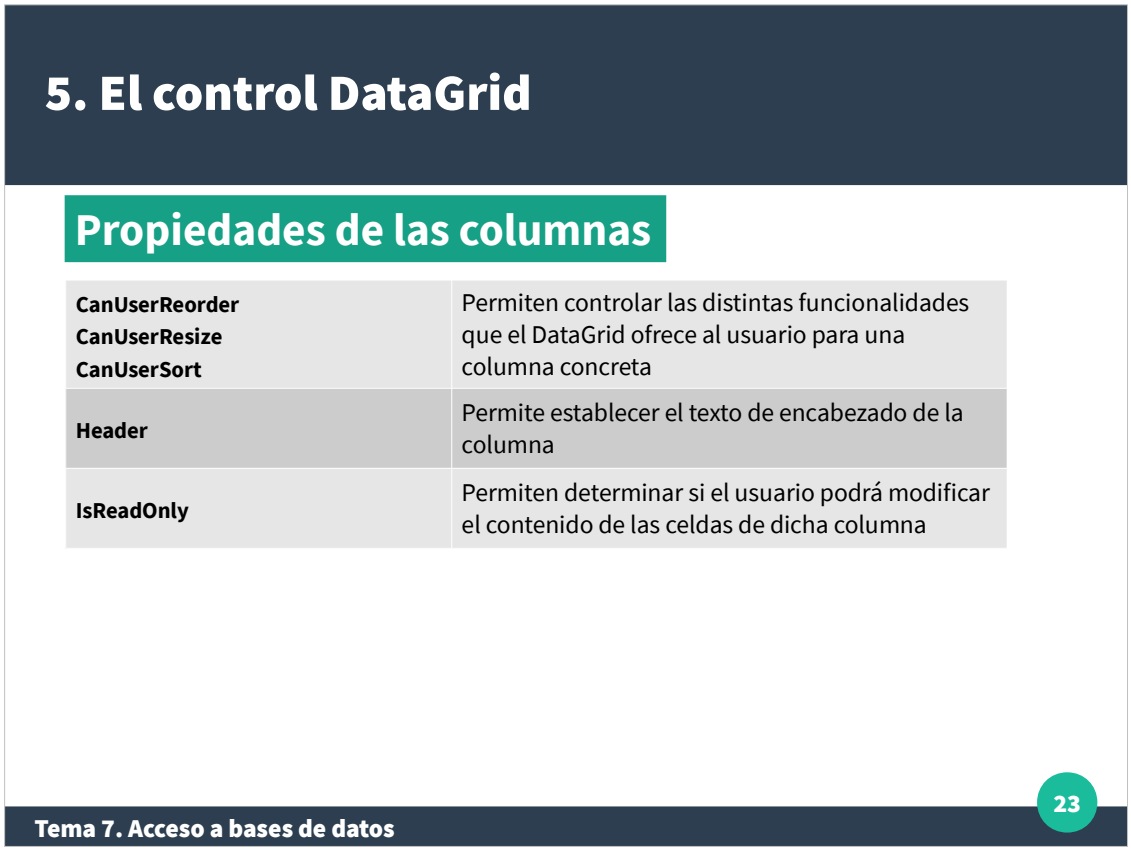
Existe un último tipo de columna, las *TemplateColumn*, que permiten definir una plantilla de datos asociada a la celda, de forma similar a como hacemos con los elementos de un control de lista.



El control *DataGrid* dispone de muchas propiedades para personalizar su funcionamiento. Podemos controlar cómo el usuario seleccionará los elementos del *DataGrid* (si selecciona siempre filas completas o puede seleccionar celdas aisladas, por ejemplo).

También podemos personalizar las funcionalidades que el *DataGrid* ofrece al usuario: ordenación de columnas, cambiar el tamaño de filas y columnas, añadir y eliminar filas,…

También podemos controlar el aspecto visual del *DataGrid*, por ejemplo añadiendo un efecto pijama (color de fondo diferente para filas alternas) o estableciendo un estilo a diferentes niveles (encabezados de filas o columnas, filas completas o celdas).

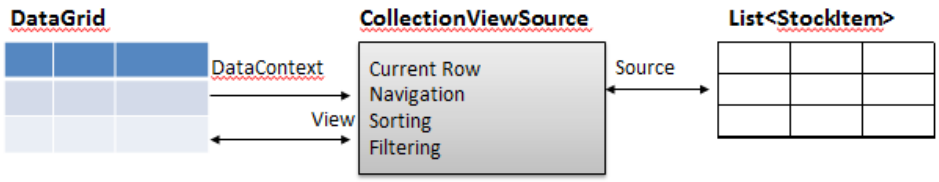


Además de las propiedades generales del *DataGrid*, cada una de sus columnas también tiene una seria de propiedades para personalizarlas, además de las utilizadas hasta ahora (*Header* y *Binding*).

Por ejemplo, podemos controlar si la columna ofrece la funcionalidad de ordenación o de redimensionado, o si el usuario podrá modificar los datos de dicha columna.

5. El control DataGrid

Filtro de datos




```
//Creamos una nueva vista
CollectionViewSource vista = new CollectionViewSource();

//El origen de la vista es la ObservableCollection
vista.Source = contexto.PEDIDOS.Local;

//El DataContext del DataGrid es ahora la vista
PedidosDataGrid.DataContext = vista;
```


Aunque como hemos visto es posible enlazar directamente un *DataGrid* a una lista de elementos, es muy común utilizar un objeto intermedio de tipo *CollectionViewSource*. Este objeto ofrece capacidades de ordenación, filtrado y agrupamiento de registros, además de mantener información del registro actual.

Para utilizarlo, como se puede apreciar en el ejemplo, se asocia a la lista de registros mediante la propiedad *Source*. Además, se modifica el *DataContext* del *DataGrid* para que sea la vista, en lugar de la lista de registros.

Una de las posibilidades que ofrece el *CollectionViewSource* es la de filtrar los datos que se muestran en el *DataGrid*.

Para ello, será necesario definir un manejador para el evento *Filter*. Cuando la vista se refresque (mediante una llamada a su método *Refresh*) el manejador se invocará una vez por cada item presente en la lista de origen. Desde el manejador, tenemos acceso al item a través del parámetro *e*.

En el código del manejador, aplicaremos la lógica de filtrado, y si el item en cuestión tiene que salir en el resultado del filtro deberemos establecer la propiedad *e.Accepted* a true. En caso contrario, la estableceremos a false.